T-SMC/19/4//27239

The Connection Machine Model CM-1 Architecture

Brewster A. Kahle
W. Daniel Hillis

Reprinted from
IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS
Vol. 19, No. 4, July/August 1989

The Connection Machine Model CM-1 Architecture

BREWSTER A. KAHLE AND W. DANIEL HILLIS

Abstract —**Massively parallel computer systems offer substantially improved performance over conventional supercomputers for many applications. Designed with many thousands of processing elements operating in parallel, these machines offer a high level of efficiency for computation-intensive algorithms. Machine architectures differ with regard to storage capacities and bandwidths, and these elements, along with software overhead, determine performance. The Connection Machine Model CM-1, is presented, a massively parallel computer that exploits data parallelism for its performance. The performance of the CM-1 system is examined with the use of a simple application. Special consideration is given to the effect of problem size and data word size on performance.**

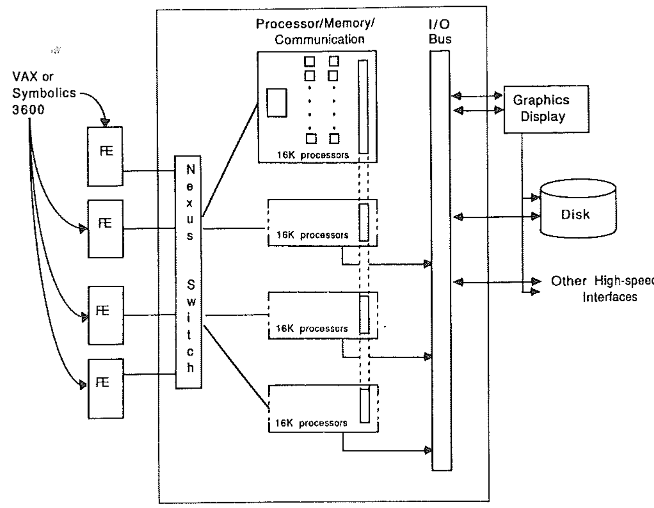

Fig. 1. Connection Machine system.

I. Introduction

THE PURPOSE of this paper is to examine the performance of the Connection Machine® Model CM-1 system, a general-purpose computer capable of adapting to many sizes and types of data sets. It will provide an overview of the hardware and software systems and present a simple application from which the raw performance of the machine can be examined. This will illustrate the power of data level parallelism and indicate how CM-1 performance is enhanced by the use of virtual processors and variable word size.

The Connection Machine system achieves its speed by operating on many thousands of data objects in parallel, rather than looping over them serially. The Connection Machine processors work with a conventional serial front-end computer as a controller, which issues the instructions executed by the Connection Machine computer. This paper describes the first commercial implementation of Connection Machine architecture, the Thinking Machines Corporation Model CM-1.

Other computers have been proposed and built that have varying similarities to the CM-1 architecture. Some machines differ in their control, such as the BBN Butterfly [1] and the NYU Ultracomputer [2]. Other machines differ in their intercommunication networks, such as the Massively Parallel Processor [6], Non-Von [8], Dado [9], and ICL DAP [10]. Most of the ideas in the CM-1 system have their roots in the long history of both parallel and serial processors that have been proposed and built. The similarities and differences with these machines will not be covered in this paper.

II. Hardware Overview

The CM-1 system consists of a processor array, from one to four front-end computers, and high-speed peripherals such as disks and image devices (see Fig. 1). The processor array contains 64000 processing elements, each a simple serial processor with 4000 bits of memory giving 32 Mbytes of total memory for the machine. A full CM-1 system is made up of four sections of 16000 processors each. The sections can be used separately, in pairs, or as one 64 kbyte processor unit.

Aside from its processors, each section contains interprocessor communication hardware, a sequencer, and, optionally, a set of peripherals. The interprocessor communication hardware is used for communication between any processors controlled by one host. A sequencer receives macroinstructions from the front end and broadcasts sequences of microinstructions to all the processors in its section. A typical macroinstruction would be to add two 32-bit numbers and store them in a particular location. Because of the simplicity of the processors, each macroinstruction is typically implemented by many microinstructions.

The front end computer, or host, attaches to the microcontroller through a bidirectional crossbar switch called a

Manuscript received April 27, 1987; revised October 31, 1987.
The authors are with Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1280.
IEEE Log Number 8427235.
®Connection Machine is a Registered service mark of Thinking Machines Corporation.

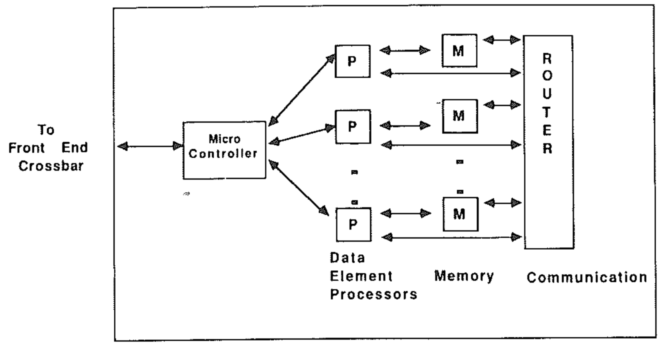

Fig. 2. Processor/memory/communication system.

Nexus, and control one, two, or four 16-kbyte processor sections. Typical front ends are the VAX® or the Symbolics 3600® Series computer.

Although the front end can be used to supply data to the CM-1 computer, in many applications the CM-1 processors can process data much faster than the front end can supply it. For this reason, the CM-1 processors are connected to a high-speed bidirectional bus consisting of up to eight I/O channels, each with a bandwidth of 320 Mbits/s. Special disk drives, frame buffers, frame grabbers, and specialized I/O devices can effectively use this bandwidth. This large I/O bandwidth makes the machine effective in very large data-intensive problems such as data bases, image analysis, and graphics.

Pointer referencing (or referencing the data in one data object from another) on the CM-1 computer requires interprocessor communication. Since a CM-1 processor needs to be able to access data from any other processor, this intercommunication system has to handle a large load at high speed. This is achieved by the use of a router which is integrated into the architecture so that every processor's memory is easily accessible to every other processor. The result is that the application programmer does not have to worry about physical processor geometry since the router handles all interprocessor communication efficiently regardless of layout (see Fig. 2).

Along with general pointer referencing, two-dimensional interprocessor communication is supported. In this type of communication, each data object can communicate with its two-dimensional neighbor (to the north, east, west, or south). This intercommunication is handled by a slightly different mechanism and is faster than using the general communication mechanism. Image processing applications often uses this ability to move data from one pixel to the next. The performance of these individual elements will be treated later in this paper.

III. Software Model

From a programming point of view, the CM-1 system can be viewed as a smart memory that can be addressed and controlled by a front-end computer. The control of the program lies solely in the front end, while data are distributed between the front end and the CM-1 computer, depending on where it can be operated on most efficiently.

This section is intended as an introduction to programming the CM-1 computer. It will describe how data map onto the CM-1 processors and the types of operations that can be performed on CM-1 data. Finally, this section will present an example application and show how it can be implemented on the CM-1 system.

A. The CM-1 Computer as Smart Memory

At the heart of any large problem is the data set consisting of some combination of interconnected data objects such as numbers, characters, records, structures, and arrays. In any application these data must be selected, combined, and operated on. Data parallelism takes advantage of the parallelism inherent in large data sets.

To make effective use of data parallelism, the CM-1 computer has to be made to "look" like the data of the problem being solved. To do this, the processors in the machine are each assigned one data object, which can be any combination of immediate data and pointers. Operations can be specified to operate simultaneously on any or all data objects in the machine.

The CM-1 processors should be used whenever an operation can be performed simultaneously on many data objects. Data objects are left in the CM-1 computer during execution of the program and are operated on in parallel at the command of the front end. This differs from the serial model of reading each data object from its memory one at a time, operating on it, and then storing it again.

The front-end computer handles the flow of control, storage, and execution of the program, and all interaction with the user and/or programmer. The data set, for the most part, is stored on the CM-1 computer. The front end can fetch and store memory in individual processors efficiently to perform serial operations if this is desirable.

There are several direct benefits to maintaining program control only on the front end. First, programmers can work in an environment familiar to them. Programming languages, programming concepts, and program development support (compilers, debuggers, editors, etc.) do not change. Second, most of the existing code for an application does not have to be changed to use the CM-1 processors. A large part of most application programs pertains to the interface between the program, the user, and the operating system. Since the control of the program remains on the front end, code developed for these purposes is useful with or without the CM-1 computer, and only the code pertaining specifically to the data residing on the Connection Machine computer needs to be coded to use the CM-1 processors. Parts of the program which are especially suited for the front end, such as file manipulation, user interface, and serial data operations, can be done on the front end, while the parts of the program that run efficiently on the CM-1, namely the "inner loop" that operates on the data set, can be done there. In this way, the individual strengths of both the serial front end and Connection Machine computer can be maximized.

®VAX is a Registered service mark of Digital Equipment Corporation.
®Symbolics 3600 is a Registered service mark of Symbolics, Inc.

Most large problems have a large data set composed of interconnected structures, arrays, and records which can be mapped directly onto CM-1 processors. *Each processor's memory in the CM-1 computer holds the data from one record or array element and some small stack space for temporary results.* A processor can hold data of different types such as integers, floating point numbers, symbols, or pointers. Typically, each processor only needs 128–1024 bits for this information. Rarely is running out of memory within one processor a problem because each physical processor contains 4000 bits of memory.

Problems of different sizes will use different numbers of processors. Some problems will allocate and deallocate processors during the course of running the program. Because the CM-1's flexibility this works well. If a program needs fewer than 64000 processors, then some sit idle, ignoring the instructions being sent by the front end. If a program needs more assigned processors than 64000, then many more "virtual processors" are available (see Section VI).

Examples of data objects used in the CM-1 computer and pixels, transistors, data base records, atoms, and sections of physical space. Each of these are allocated one per processor. (Throughout this paper the terms "processor," "data object," and "record," are used interchangeably.)

B. Program Example

Consider the problem of manipulating a data base of employee records where each employee is a structure of employee ID, title, salary, a pointer to their boss, and a pointer to their assistant (if they have one). The example operations are "give each manager that earns over \$100000 a pay cut of \$5000," give all managers' assistants a ten percent raise," and "sum all the managers' salaries" to illustrate selection and parallel operation, pointer reference, and global operations respectively.

For this sample problem, each employee's record would be assigned to a processor. On a serial machine, data are loaded from a file and put into records in the virtual memory of the serial machine. On the CM-1 system, the difference is the records are kept on the CM-1 computer.

The language *Lisp® pronounced star-lisp) is used in this paper to describe the pieces of code for the CM-1 computer. All examples can run on the CM-1 computer. In *Lisp the employee structure might look like this:

```
(*defstruct (employee))
   employee-ID;   type: number
   position;      type: number
   salary;        type: number
   boss;          type: pointer
   assistant;     type: pointer.
```

This declaration indicates that structures of this type should be created and manipulated in the CM-1 processors. If the creation and manipulation were done with the serial front end, then these operations would take about as long using the CM-1 processors as using the serial machine's own virtual memory. Thus using the CM-1 memory from the serial machine as if it were its own is not a very significant speed penalty even if one never used the Connection Machine processors.

The data in each processor (in this case, an employee's record) will be a set of fields of numbers and pointers. That a pointer is an address of another processor just means that the program can get or send data to that other processor; this is called moving data through a pointer, or pointer reference. So far, nothing is different in the way the CM-1 computer feels or performs from programming the front end. In fact most programs look very much the same whether the data are on the CM-1 computer or not (see Fig. 3).

Consider the first operation, to select all managers that earn over \$100000 and decrease their salary \$5000. To execute this we must select all the data objects that satisfy the predicate "is a manager and makes over \$100000." This selection is done by instructing each processor to be "active" when the value in the employee-position slot is equal to *manager* and the value in the employee-salary slot is greater than \$100000. Notice that *manager* is a value known by the front-end and is broadcast to each processor to compare against its local data. In *Lisp the front-end code might look like this:

```
(*when (and (= !! position (!! *manager*))
            (> !! employee-salary 100000)))
   (*set employee-salary (-!! employee-salary (!! 5000))).
```

Executing this code on the front end would deliver an instruction stream that would look something like:

```
(let ((saved-context-bit *stack-index*))
   (CM:store-context-always saved-context-bit)
   (CM:u= constant (pvar-location position) *manager*
    8)
   (CM:and-context-with-test)
   (CM:u> constant (pvar-location employee-salary)
    100000 20)
   (CM:and-context-with-test)
   (CM:u-constant (pvar-location employee-salary) 5000
    20)
   (CM:load-context-always saved-context-bit)).
```

Notice that these instructions look like a serial computer's macroinstruction set. We have selected a set of processors and executed an operation within those processors.

To perform the "give each managers' assistant a ten-percent raise" instruction requires communication between processors. This is done by referencing data in the processor pointed to by a field in another processor. In other words, a processor would have an address (pointer) in one of its slots to another processor.

This operation is done in the following steps:

- each manager's processor gets the salary information from their assistant (in the program this is "pref!!"; or parallel reference);

®*Lisp is a Trademark of Thinking Machines Corporation.

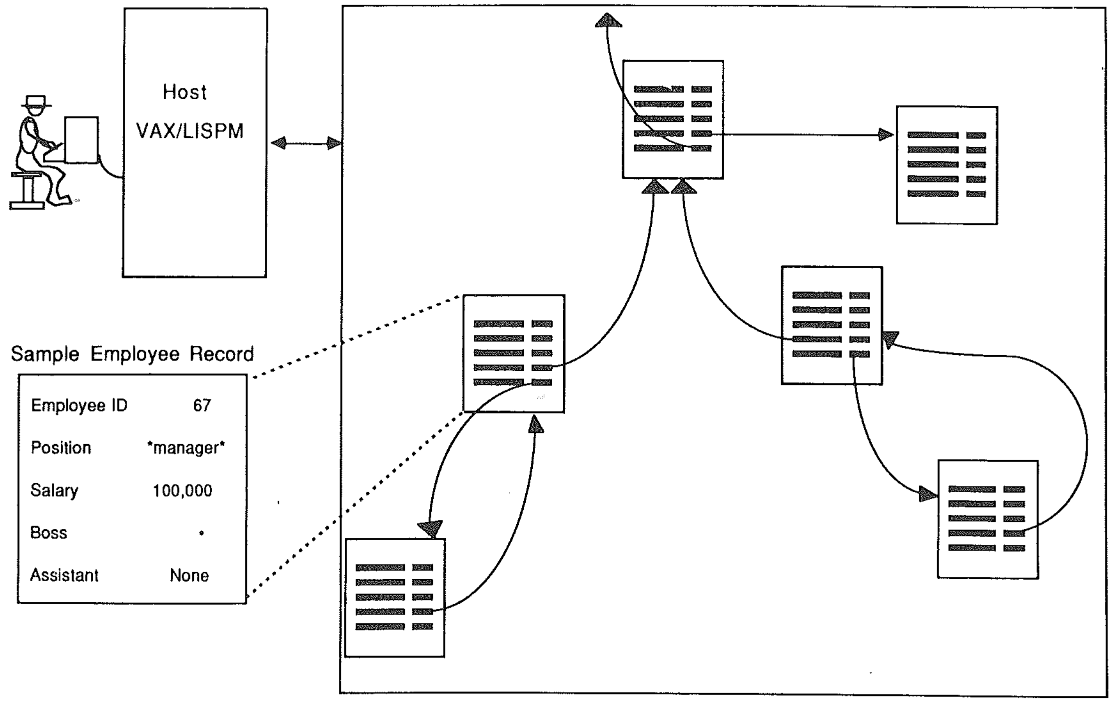

Fig. 3. Connection Machine processors: software view.

- the manager's processor saves that information in a temporary location (this is a move with the processor to the stack (the "*let"));
- the new salary is computed by multiplying the old one by 1.10 in each manager's slot (the "*!!");
- the result is sent back to the assistant's processor and put in the salary slot. This involves another move through a pointer. (This is the pset, or memory parallel set.)

This operation is done, in *Lisp, as

```
(*let (assistants-salary!! (pref!! assistant employee-
  salary))
  (*pset assistant employee-salary
    (*!! assistant-salary!! (!!0.10)))).
```

The operation, "total all managers' salary," returns a number to the front-end machine which is the total of all managers' salary slots. This is done by selecting all managers and then doing a *global sum*. Global sum is implemented using a logarithmic time tree reduction using the intercommunication wires directly. In *Lisp this looks like:

```
(*when ( =!! position (!! *manager*))   ;select managers
(*sum employee-salary))                 ;return sum.
```

This is an example of the use of *scan* that is discussed in the next section. This example shows how a specific example might map onto the Connection Machine processors and how common operations are used on such data.

IV. Basic Performance of the CM-1 System

The performance of the CM-1 system depends on the relative speeds of different common operations. Each application uses a different mix of instructions, so exact application performance depends on processor utilization and the weighted average of the instructions used. The performance rates shown in Table I describe 32-bit operations from the point of view of an active processor as a ratio to the speed of an add. Further, this section describes how these numbers scale with different arithmetic precisions and "virtual processor" ratios.

TABLE I
Relative Times of Various 32-Bit Operations

Add	1
Move within processor	0.6
Multiply	32
Move with nearest grid neighbor	4
Move through pointer (pointer reference)	25
I/O move (our or into the CM-1)	20
Scan	25
Global mix	2
Global sum	25

Performance per processor is the operation speed of an active processor or data object. A 32-bit integer addition, for instance, takes about 32 μs per processor. The system performance depends on both the speed per processor and the number of active processors. Thus the peak system performance in additions is about 2000 million instructions per second (MIPS). Other elements that influence performance are the instruction mix used by the application, the precision of the arithmetic and the total number of processors used. Some applications, such as image processing and cellular automata, run significantly faster than 2000 MIPS because they commonly use instructions on less than 32 bits. Table I uses 32 bits as a unit of measurement for other performances.

Table I describes the machine as a set of ratios with add time. Each entry is described in the following.

The 32-bit add is the unit in Table I. An add is a typical logical operation where the operands and results stay

within each processor. Each of these fields might be on the stack or a slot in the data object. Other operations having similar speeds are logical operations such as OR, AND, XOR, etc. Also, if one of the operands to these instructions is an "immediate" number (a literal number that comes from the instruction stream rather than from local processor memory) the time is about the same.

The move within a processor command moves a field from one location to another within each processor. An example of this is moving an argument to the stack.

Multiply is implemented with a "shift-add" for each bit, so it takes 32 times longer than each add because it is a 32-bit multiply. This corresponds to the *!! in the program example.

Move with nearest grid neighbor are transfers of 32 bits with the processor to the "east," "west," "north," and "south." All active processors will do this transfer at once. This operation is useful in image processing and graphics applications.

Move through a pointer is a reference into another processor's memory. This corresponds to the pref!! (get the assistants' salaries) in the program example. The performance of this operation varies only slightly with the layout of the processors. Of course, if the problem is a grid problem, then the grid communications should be used instead.

I/O move is the time required to get the I/O port on the CM-1 computer. The real I/O performance will depend on the I/O device being used because most will not sustain a 320-Mbit/s/8000 processor rate. The important aspect in this performance is its balance with the other performances so that the CM-1 computer can be connected directly into data-intensive applications.

Scan is an operation that performs a form of processor intercommunication very quickly. Add-scan, for instance, results in each processor getting the sum in all processors before it. This is implemented with a parallel algorithm. Max-scan leaves the maximum up to that point:

Processor number	0	1	2	3	4	5	6	7	8	9	10
Value	0	1	1	0	0	1	1	2	1	0	1
Result of add-scan	0	1	2	2	2	3	4	6	7	7	8
Result of max-scan	0	1	1	1	1	1	1	2	2	2	2.

This is a useful operation on the CM-1 computer for implementing other instructions such as global-sum but many uses have little analogy with serial software cliches.

Global-max results in the maximum value held by any active processor being passed back to the front end computer. The speed of this instruction is due to special hardware. Executing an AND of a field in all processors uses the same mechanism and runs at approximately the same speed. Similarly, OR and MINIMUM perform about the same.

Global-sum is the sum of all values held by processors in the CM-1 computer. This is implemented using *add–scan* and taking the highest processor's result.

TABLE II
RELATIVE TIMES FOR OPERATIONS ON N-BIT NUMBERS

Add	N
Move within processor	N
Multiply	N^2
Move with nearest grid neighbor	N
Move through pointer (pointer reference)	N
I/O move (out or into the CM-1)	N
Scan	N
Global max	N
Global sum	N

V. PERFORMANCES WITH DIFFERENT WORD LENGTHS

The effective performance will change based on what precision arithmetic is needed. "Word length" is used to describe the precision of the arguments to the CM-1 instructions. The previous section assumed 32-bit arithmetic for each of the operations. The Connection Machine system can operate on a few bits up to hundred of bits with the same ease or programming. Variable length arguments are handled in microcode for efficiency and ease of programming. The language *Lisp, for example, can extend the word length of numbers as needed. Sometimes a program will need only 16-bit numbers to start, but will use 32 or 64-bit numbers later (or 53 bits for that matter). It is, of course, desirable to minimize the precision required to maximize performance and minimize space utilization.

As one can see in Table II, most operations scale linearly with the word length. Thus if only 16 bit additions are needed, twice as many can be done in the time of a 32-bit addition. There is a small amount of calling overhead in each instruction call, so behavior is not quite linear for very short numbers.

Multiplication uses a shift–add algorithm so that the precision scales with the square of the number of bits; thus a 16-bit multiply would take 1/4 the time of a 32-bit multiply. The CM-1 system's flexibility extends to the precision of the arithmetic needed. Image processing applications often use very limited precision numbers, and numerical simulation often use very large precision numbers.

VI. VIRTUAL PROCESSOR CONCEPT

As presented so far, the CM-1 system has a constant (independent of problem size) performance of an add operation for problem sizes up to 64000 processors. In real applications, often 64000 processors are not enough. For example, an image processing application with a 1000 × 1000 image would require one million processors, one for each pixel. To get around the arbitrary limit of the number of physical processors, the CM-1 supports virtual processors (see Fig. 4). Virtual processors are a software abstraction, implemented at the microcode level, which allows a programmer to write programs that are independent of the number of physical processors that the CM-1 hardware contains. This section will describe how virtual processors are implemented on the CM-1, and what effect they have on program performance.

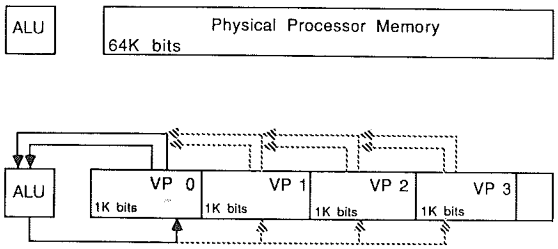

Fig. 4. Virtual processor (VP) concept.

Virtual processors are implemented using three separate mechanisms: one for storage, one for processing, and another for communications. First, the memory of each physical processor is divided evenly among the virtual processors assigned to it. The number of virtual processors per physical processor is referred to as the virtual processor ratio. For example, a problem requiring 1M (2^{20}) virtual processors would need a virtual processor ratio of 16 to operate on a 64000-machine ($2^{20}/2^{16} = 2^4 = 16$). This means that the physical address space of each virtual processor would be divided into 16 equal address spaces, one for each virtual processor assigned to it. Image processing applications commonly use only 100 bit/virtual processor so 32 processors can be assigned to each physical processor, and the CM-1 computer can be configured as a two million-processor machine.

The second mechanism necessary to support virtual processors is time-multiplexing of the physical processors among the virtual processors assigned to it. Every macroinstruction sent by the front end is run on each of the virtual processors within each physical processor. The overhead for switching context is extremely small (about the time to execute a 2-bit add) because each processor is so simple. In the 1M processor example given above, each macro instruction would be executed 16 times by each physical processor.

The third mechanism of communications allow the CM-1 processors to communicate with one another without regard to virtual processors. Grid communications are handled by the microcode by sharing the grid wires. General communications (pointer reference) is handled by the router hardware. The length of a processor address changes based on the number of virtual processors in the machine. This virtual address is used by the router hardware to deliver messages to the correct virtual processor.

Since the virtual processor abstraction is implemented at the microcode level, the virtual processing mechanism is both efficient and totally transparent to the programmer. These characteristics make the CM-1 system appropriate for problems of very different sizes. In general, it is not necessary to know how many physical processors are in a CM-1 computer.

A. Effect of Virtual Processors on Performance

Since a physical processor is shared among several virtual processors, the performance from a virtual processor's point of view goes down. If one measures the performance of the CM-1 computer as a whole, however, in terms of MIPS or MFLOPS, the performance does not go down but remains constant. In other words, given a virtual processor ratio of 16, each virtual processor is only getting 1/16 of the physical processors time, so that each macroinstruction takes 16 times as long, but each macroinstruction is doing 16 times as much work as in the case of a virtual processor ratio of 1.

TABLE III
RELATIVE TIME OF OPERATIONS WITH K VIRTUAL PROCESSORS ($K = 1000$)

Add	K
Move within processor	K
Multiply	K
Move with nearest grid neighbor	K
Move through pointer (pointer reference)	$K \log K$
I/O move (out or into the CM-1)	K
Scan	K
Global max	K
Global sum	K

Table III shows the effects of virtual processors on the performance of the various classes of macroinstructions. As expected, most operations slow down linearly with the virtual processor ratio. The only exceptions are those that involve interprocessor communication which slow by $O(K \log(K))$ for virtual processor ratios less than 16. The performance for very high ratios has not been studied. The addition of the $\log(K)$ term is due to the increase in router congestion.

Virtual processors free the programmer from the physical size of the CM-1, and also from the size of the data set that the program is to operate on. Since this is done at a very low cost in speed and programming, virtual processors have been used in most applications programmed for the CM-1 system.

VII. Conclusion

The model CM-1 system offers a high level of performance due to its flexible massively parallel architecture. Much of the maximum peak performance of 2000 MIPS can be used on a problem due to the balanced performances of all common instructions. The CM-1 can be used for very different types of applications because the machine can be efficiently reconfigured to have the needed number of processors and the needed arithmetic precision.

Since the CM-1 computer works closely with a serial front end computer, the programmer operates in a familiar environment. The CM-1 system combines the benefits of programming the front end computer with the performance of a massively parallel machine.

References

[1] Bolt Beranek and Newman, Inc., "Development of a butterfly multiprocessor test bed," Rep. 5872, Quarterly Tech. Rep. 1, Mar. 1985.

[2] A. Gottlieb *et al.*, "The NYU ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, pp. 175–189, Feb. 1983.

[3] W. D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.
[4] "Introduction to Data Level Parallelism," Thinking Machines Tech. Rep. 86.14, Apr. 1986.
[5] C. Lasser and S. Omohundro, "The essential *Lisp manual," Thinking Machines Corp., July 1986.
[6] D. H. Schaefer, J. R. Fischer, and K. R. Wallgren "The massively parallel processor," *Eng. Notes*, vol. 5, no. 3, pp. 313–315, May–June 1982.
[7] H. J. Siegel *et al.*, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, no. 12, Dec. 1981.
[8] D. E. Shaw, "The NON-VON Supercomputer," Dep. Comput. Sci., Columbia Univ., New York, NY, Aug. 1982.
[9] S. J. Stolfo and D. E. Shaw, "DADO: A tree-structured machine architecture for production systems," Dep. Comput. Sci., Columbia Univ., New York, NY, Mar. 1982.
[10] P. M. Flanders *et al.*, "Efficient high speed computing with the distributed array processor," in *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. New York: Academic, 1977, pp. 113–127.

Brewster A. Kahle is a Scientist at Thinking Machines Corporation, Cambridge, MA. He is currently working on architectures and applications for the Connection Machine.

W. Daniel Hillis is a Founding Scientist of Thinking Machines Corporation, Cambridge, MA, currently working on machine learning and parallel computing. He is the principal architect of the Connection Machine computer.

Mr. Hillis won a 1985 ACM Distinguished Thesis Award for the book *The Connection Machine*.